Thesis for The Degree of Doctor of Philosophy

Once More, With Combinators: Designing a Low-Power Architecture for Functional Programming

Jeremy Matthew Gardiner Pope

Division of Computing Science
Department of Computer Science & Engineering
Chalmers University of Technology | University of Gothenburg
Göteborg, Sweden, 2024

Once More, With Combinators: Designing a Low-Power Architecture for Functional Programming

Jeremy Matthew Gardiner Pope

ISBN 978-91-8103-062-4
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 5520
ISSN 0346-718X

Department of Computer Science & Engineering
Division of Computing Science
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2024.

Q X I # V R E L H A M G
S U I L E B I S J T O S
P N L L U R S O T O F U
E I B A N D O R T U S N
X L H C A B N N O O M G

Abstract

The Internet of Things (IoT) consists of a growing number of networked appliances, from coffee machines to door locks. Each of them contains one—or more—microprocessors responsible for their operation and communication. While some appliances may have an abundance of resources, others need to operate under significant energy and size constraints, encouraging the use of low-power microcontrollers.

Unfortunately, many of these microcontrollers are programmed in low-level languages due to their constraints, resulting in security vulnerabilities. These are worsened by the physical presence that IoT devices have: as appliances, they can usually interact with the environment through sensors and actuators, making vulnerabilities particularly concerning from privacy and safety perspectives. Lazy, pure functional programming languages are promising both in ease of development and correctness, but have historically been difficult to run without considerable resources.

This thesis aims to bring these languages to IoT devices through the creation of a low-power processor architecture, Cephalopode, that carries out graph reduction in hardware. Synthesis and simulation of Cephalopode indicates that it significantly outperforms a combinator-based software implementation on a RISC-V core, and suggests that it would match (and sometimes exceed) the performance of GHC if the latter were able to emit code for RISC-V. It is in turn outperformed by native C code with fixed-width integers, but nears the performance of C with an arbitrary-precision library. While designing Cephalopode several difficulties with hardware description languages were identified, leading to the development of Stately, an editor for a simple but powerful extension of finite state machines, and Bifröst, a high-level language for modular hardware design. The latter provides a novel approach to input-output that has proven to be both at a comfortable level of abstraction and suitable for use in realistic hardware designs.

Keywords

Functional Programming, Internet of Things, Architectures, Hardware Description Languages, High-Level Synthesis

Acknowledgments

This work was partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation and by a grant from the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023).

I would like to thank my supervisor Carl Seger for his encouragement, patience, gusto, and sharing of a wealth of uncommon knowledge, and my co-supervisor Mary Sheeran for her invariably wise insights and guidance toward stronger research foundations. I would also like to thank Henrik Valter for his invaluable contributions to Cephalopode and its evaluation, and our visiting students from France—Jules Saget, Nicolas Nardino, and Dorian Lesbre—for their various contributions to the work described in this thesis.

Thanks goes to all of my pals at CSE, including but not limited to: the original office riff-raff Agustín, Nachi, and Matti; my fellow hot chocolate afficionado Iulia; dom goa gubbarna Benjamin o Alexander; the new kids on the block Abhiroop and Robert; the wizard Francisco; the well-typed Carlos; the gracious Mohammad; the VLSI survivor Henrik; the mischievous Hannaneh; the cultured minister of cappuccino, computing, and philosophy Ivan; and the venerably awesome Irene.

A great proportion of any knowledge and passion of mine I owe to wonderful teachers I had during school, and I am eternally grateful to them for inspiring and nurturing my interests in computer science, mathematics, and language.

Finally, I give my thanks and endless love to my partner Julia, my dear old friends Anastasiya, Lawrence, and Daniel, my best brother Geoff and sister Lizzy, my fantastic parents Janet and Norris, the esteemed Gilbert, and my other friends and family; old and new, art and science, day and night, near and far, you all make things wonderful.

List of Publications

Appended publications

This thesis is based on the following publications:

1. Jeremy Pope, Carl-Johan H. Seger, Henrik Valter "Higher-order Hardware: Implementation and Evaluation of the Cephalopode Graph Reduction Processor"
   Under submission.

2. Jeremy Pope, Jules Saget, Carl-Johan H. Seger "Cephalopode: A Custom Processor Aimed at Functional Language Execution for IoT Devices"
   MEMOCODE, 2020.

3. Jeremy Pope, Jules Saget, Carl-Johan H. Seger "Stately: An FSM Design Tool"
   MEMOCODE, 2020.

4. Jeremy Pope, Carl-Johan H. Seger "Bifröst: Creating Hardware With Building Blocks"
   FDL, 2023.

Research Contribution

For papers A and B, the Cephalopode architecture was jointly designed by myself and Carl Seger, and evaluated largely by Henrik Valter. In both versions I was responsible for the high-level design of the graph traversal process, snapshot memory, and garbage collection, and implemented parts of the reduction engine. For the newer version (presented in Paper A), I also implemented the control unit of the garbage collector and several other small parts of the processor.

For Paper C, I did the majority of the design work for Stately (with input and feedback from Carl Seger and Jules Saget), and all of its implementation.

For Paper D and Chapter 6, I designed the Bifröst language with substantial input and feedback from Carl Seger and Dorian Lesbre, and implemented its compiler.

Contents

Chapter 1

Introduction

1.1 The Internet of Things

The Internet of Things, or IoT, describes the emerging network of internet-connected appliances, ranging from smart light bulbs to humidity sensors to coffee machines. A unifying feature is that the devices are appliances rather than general-purpose computers (as far as the user is concerned, at least): they fulfill specialized roles and functions, as opposed to offering the broad functionality of ordinary computers. This specialization typically involves more rich interaction with the physical world, which takes place through sensors (e.g. a thermometer) and actuators (e.g. a heating element).

Although some IoT devices have the luxury of ample resources, many are constrained in size, cost, and energy. Consider for example a smart light bulb: the size of the housing, electricity cost to the user, and unit price all provide constraints on the hardware within. Industry goals include an even more dramatic constraint, namely being able to power a device from one battery for ten years [1], and at least one company sells wireless sensors powered entirely from harvesting energy in indoor environments [2]. Although not suited for that degree of low energy, many IoT devices contain low-power microcontrollers or systems-on-a-chip (SoC), which offer limited speed and memory compared to the hardware found in a desktop computer or laptop. On the lower-power variants, running a conventional operating system is out of the question in terms of both memory and speed.

These low-power IoT platforms are typically programmed in a low-level language—often C—both for the sake of efficiency, direct access to resources such as I/O pins, and ubiquity. Such a programming environment poses two serious problems: first, the difficulty and inefficiency of programming without access to the abstractions that are routinely used in higher-level programming languages, and second, the tendency to introduce dangerous bugs due to the simultaneous complexity of the programming model and lack of safety provided by the language. Memory safety is one such mechanism commonly lacking, making it easy to inadvertently write to an invalid or unexpected location in memory (dubbed *memory corruption*), in turn leading to wildly erroneous behavior. A more subtle danger is that of *integer overflow*, where arithmetic on fixed-size data produces a result outside of what can be represented in the fixed number of bits it occupies, silently resulting in a partial, incorrect value. This can happen in intermediate values inside of larger computations, even if the

final result would be within a safe range—for example, the sum overflowing when computing the mean of a set of numbers, even though the mean itself is certainly not too large to store—and even sub-expressions in a single line of code, making this type of error particularly easy to make. It is worth noting that these types of errors can also combine: reasoning about memory addresses using arithmetic that can fail due to overflow can in turn lead to writing to unintended areas of memory, for example as seen in a glibc format string vulnerability [3]. Despite being well-known for many years, both memory corruption and integer overflow bugs have a long and distinguished history of exploitation that continues through the present day.

Given also the networked nature of IoT devices, it is perhaps unsurprising that security vulnerabilities abound—a recent example being the combination command injection and buffer overflow attack on the widely-used ThroughTek Kalay platform [4]. The capability of IoT devices to interact with the physical world makes this especially concerning, as it introduces novel opportunities for attackers; in the case of the previous example, complete control over a smart home camera.

These concerns are not lost on academia, industry, or hobbyists; a wide variety of approaches are being investigated toward mitigating these risks. IoT-oriented operating systems like Zephyr [5], Mbed [6], and RIOT [7] provide some abstraction from the underlying hardware as well as complex functionality such as cryptographic support, networking facilities, and thread management. Since 2021, RIOT offers support for the Rust programming language [8], an exciting development due to Rust's combination of high-level features, safety, and suitability for systems programming. Zephyr and Mbed are both designed for applications to be written in C or C++, although one user [9] managed to use a limited form of Rust with Zephyr via transcompilation to C. MicroPython [10] brings the Python programming language to microcontrollers, another step toward high-level programming of IoT.

There are also several projects that utilize functional languages to program microcontrollers. Juniper [11] is a functional reactive programming language for Arduino microcontrollers [12], with a compiler that generates Arduino-compatible C++ code. Copilot [13] is a Haskell EDSL for stream-based programming, also for Arduino. Haskino [14] provides an imperative Haskell EDSL for programming Arduinos. mTask [15] is an Clean EDSL for programming microcontrollers using the Task-Oriented Programming [16] paradigm. A subsequent paper [17] discusses how to compile mTask programs into bytecode suitable for microcontrollers; it appears that higher-order functions are not allowed. LispBM [18] and uLisp [19] are small implementations of LISP-like languages designed for microcontrollers. MicroHs [20]—to be discussed in more detail in later section of this thesis—is a small, stand-alone Haskell compiler that can generate code for both x86-64 and several microcontrollers. No IoT-oriented features are included, however the foreign function interface can be used to run platform-specific C code.

1.2 Pure functional programming

While there are plenty of high-level languages in other paradigms, pure functional programming is of particular note for several reasons (many discussed by Hughes [21]). First and foremost, it is highly composable: the ability to write higher-order functions and the separation of pure computation from side-effects allow code reuse and layering of abstractions with near-reckless abandon. Second, purity and referential

transparency allow programs to be reasoned about equationally, without undue consideration of operational semantics. Third, the properties above—along with a robust type system—make pure functional languages amenable to language-based security techniques such as those used in LIO [22].

If one narrows the scope to pure functional languages that are *lazy*—in short, that only evaluate sub-expressions on an as-needed basis—several more benefits emerge. Directly, programs may be written in a definitional manner rather than a prescription for control flow (effectful operations not withstanding), increasing the level of abstraction. This in turn benefits composition; managing control flow explicitly across the various parts of the program could be very cumbersome, which is avoided with lazy evaluation.

These benefits of lazy functional programming make it a desirable paradigm for programming, both in general and for IoT. It is not without drawbacks, however. An inherent one is that laziness makes the question of when a value is computed quite difficult to answer; where an expression is created and where it is eventually evaluated are not the same. A dramatic but perhaps less obvious drawback is the apparent "semantic gap" between lazy, pure, functional programs and the stateful, value-based architecture of traditional computer hardware. Efficiently implementing the former on the latter was initially elusive, and is still not a trivial matter.

1.2.1 Lambda calculus

Pure functional programming languages are based on Church's lambda calculus, which we look to for insights into evaluation. The primary way in which expressions are brought closer to a final value is β-*reduction*, which substitutes an argument into the body of a function; for example, $(\lambda x.f\,x\,x)(y)$ becomes $f\,y\,y$. Even ignoring other details such as α-equivalence and η-conversion, this is not enough to describe an algorithm for evaluation: when there are several locations within an expression that β-reduction could be applied, it does not indicate which one to reduce first.

Two orders are usually considered: *applicative order*, where function arguments are reduced fully before being substituted into a function body, and *normal order*, where arguments are substituted in prior to being evaluated. Applicative order is the one used by most programming languages (functional and otherwise), whereas normal order is the one used by lazy functional languages.

To illustrate the orders, consider two examples from lambda calculus extended with arithmetic: $(\lambda x.x \times x))(2+3)$ and $(\lambda x.(\lambda y.y))(10!)$. With applicative order reduction, the first expression would reduce to $(\lambda x.x \times x)(5)$, then 5×5, then 25. The second expression would reduce to $(\lambda x.(\lambda y.y))(3628800)$, then $\lambda y.y$. With normal order reduction, the first expression would reduce to $(2+3) \times (2+3)$, then $5 \times (2+3)$, then 5×5, then 25. The second would reduce directly to $\lambda y.y$.

Two facts become clear from these examples: *(i)* normal-order evaluation duplicates work that does need to be performed (e.g. $2+3$), and *(ii)* normal-order evaluation is "lazy" and avoids work that does not need to be performed (e.g. $10!$). The former is a significant performance concern, but it turns out the latter is much more fundamental. Consider the case where the $10!$ in the second example is instead something that reduces forever without becoming a value; an infinite loop or otherwise diverging expression: $(\lambda x.(\lambda y.y))(\textit{loop-forever})$. For this program, applicative order will never terminate since it becomes stuck evaluating *loop-forever*, whereas normal order successfully evaluates the program to $\lambda y.y$.

One might imagine that the opposite could also occur, but this is not the case. In fact, normal order evaluation is even more powerful than one might expect from the above: if there exists a sequence of reductions from an expression e to a value v, applying normal order evaluation to e will eventually result in v. In other words, if there is an answer, normal order evaluation will always find it, while applicative order may not.

1.2.2 Graph reduction

As we saw earlier, normal order evaluation can result in duplication of work due to its substitution of not-yet-evaluated arguments into function bodies. This can be quite bad—for example, an algorithm to perform exponentiation by repeated squaring will revert to the linear number of multiplications that a naïve algorithm would use, rather than the intended logarithmic number.

Thankfully, there is a remedy for this that does not require sacrificing the benefits of normal-order evaluation. If one considers the parse trees formed by expressions during the reduction process, this duplication of work manifests through multiple identical sub-trees that end up being evaluated independently. Consider for example the following program:

```
let square x = x * x
in square (2 + 3)
```

This corresponds to the lambda expression $(\lambda x.x \times x)(2 + 3)$ seen earlier. Figure 1.1 demonstrates the problem that arises with normal-order reduction carried out with expressions represented as trees.

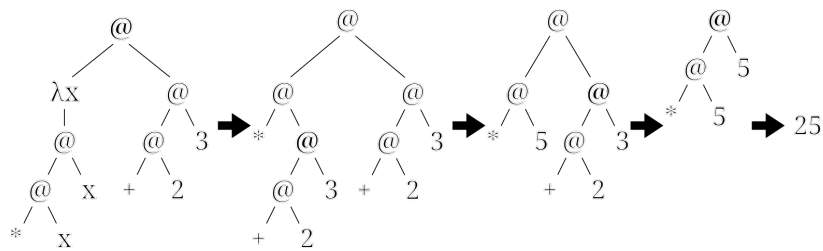

Figure 1.1: Normal order reduction with trees duplicates work (namely $2 + 3$).

Taking a cue from dynamic programming, we wish to unify these sub-trees and ensure that they are only evaluated once. This is accomplished by structuring the program as a *graph* rather than a tree, thereby allowing sub-expressions to be *shared*. In this scheme, dubbed *graph reduction*, the substitution performed in β-reduction does not copy the argument into the function body as before, but instead creates references to the argument, thereby sharing rather than duplicating it. This approach is shown in Figure 1.2.

With a caveat to be addressed in the next section, it is important that even (or especially!) when a sub-expression is shared, we still evaluate it "in place" just as before, updating it to reflect its journey toward a value: this ensures that the work going into this computation is also shared, not just the sub-expression's original form. The soundness of this last point relies on the purity of the language.

Updating (shared) expressions as they are evaluated makes sense for ones that do not require any context; for example $2 + 3$ reduces to 5 everywhere in the program,

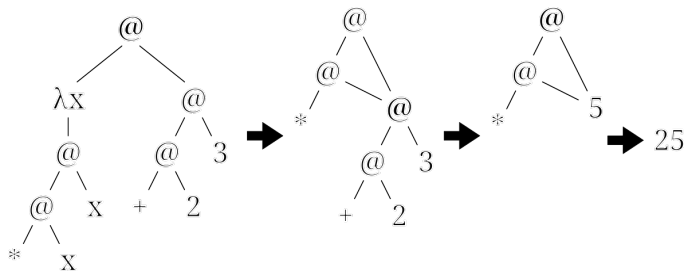

Figure 1.2: Graph reduction still operates in normal order, but avoids the duplication of the argument $2 + 3$.

and replacing it with the latter has no effect other than saving time in the future. But imagine that we encounter a function application, for example `square 7` (with the prior definition of `square`). If we carried out substitution directly on the internal structure of the `square` function's body, replacing each instance of its parameter `x` with a pointer to the argument `7`, the function would be permanently applied to `7`. Prior to sharing, this was fine, since other uses of the same function were in fact separate copies of it and thus unaffected. With sharing, however, the function itself may be applied to other arguments elsewhere, rendering this in-place substitution unsound.

Abstractly, when one considers a function $\lambda x.e$, the quantified nature of x makes its value contextual, precluding its replacement when it is shared between several contexts. The solution to this is to duplicate the function body and perform the substitution on the duplicate, leaving the original intact, a process called *instantiation*. Specifically, to reduce a function application $(\lambda x.e)(y)$ the body e is duplicated, during which each (free) x in the duplicate is replaced with a pointer to y, resulting in e'. The root of the function application is then replaced with the root of e' in some manner.

The process of instantiation requires traversing and duplicating an arbitrarily large sub-graph, which may in turn contain cycles (due to recursive functions) and references to other parts of the program that do not need to be duplicated as they do not depend on the argument. This is clearly not ideal.

1.2.3 Combinators

A solution to the difficulty of handling functions—or rather, the bound variables therein—is to simply not have them in the first place. As described by Schönfinkel and Curry, and applied to computer programming in 1979 by Turner [23], this can be achieved by introducing a handful of primitive functions called *combinators*, and translating functions defined using λ into expressions of these.

A convenient set of combinators is S, K, and I, with the following reduction rules (instead of β-reduction): $S\,x\,y\,z \Rightarrow (x\,z)(y\,z)$, $K\,x\,z \Rightarrow x$, and $I\,z \Rightarrow z$. The utility of these three is not immediately apparent. However, if one considers the expressions $S\,x\,y$, $K\,x$, and I, these can be viewed as functions that take an argument z and respectively: send it to both sides of an application, ignore it, and place it as a leaf. In other words, they can be used to "guide" an argument to anywhere inside an expression, with the restriction that the expression does not contain any

λ-abstractions.

By carrying out this step of translation recursively from the inside of an expression out, we can translate an arbitrary λ-calculus expression (i.e., a program) into one that uses these three combinators rather than any λ-abstractions. Without any λ-abstractions, there are no bound variables to worry about, and furthermore β-reduction has been replaced with a few much simpler reduction steps. As an example, the `square` function defined in the previous section would translate to the cryptic and verbose: $S\,(S\,(K\,(\times))\,I)\,I$. Applying it to an argument x results in $x \times x$ after several reduction steps.

Although sufficient, the three combinators described above are not terribly efficient: program sizes may increase quadratically [24]. Turner instead uses a larger—but still fixed—set of combinators, reducing sizes in practice. The selection of combinators and how to emit them in the compiler appears to be as much an art as a science, with MicroHs [20] and fl [25] using their own extensions of Turner's set.

The resemblance of combinators to machine instructions was not at all lost on Turner, who described a software implementation but hinted at a hardware one. Surely enough, several combinator-based processors were created in the years thereafter, and will be discussed in a later section.

1.2.4 Super-combinators

In 1982 Hughes [26] introduced super-combinators as a way to take some benefits of combinators, but avoid some of their detriments. Super-combinators are essentially combinators derived from the source program in a manner that guarantees full laziness. They have a clearer connection to the original source program, easing debugging, and perform more work than the relatively simple fixed combinators. They also have a smaller worst-case expansion, $O(n\log(n))$ [24]. They can be compiled into code for a traditional architecture in a fairly straightforward manner, but do not appear as hardware-friendly as fixed combinators due to their flexibility.

1.2.5 Abstract machines

The G-machine [27], described by Johnsson in 1984, is an abstract machine for carrying out graph reduction. It sits between the idealized view of graph reduction and a traditional computer architecture, making it relatively easy to compile a lazy language (in the paper, Lazy ML) to, and also easy to translate into machine code for a real computer (VAX-11). As well as being well-suited for traditional architectures, the G-machine makes it possible to avoid a substantial amount of intermediate bookkeeping that graph reduction normally performs; according to the paper this led to a 10x speedup for some programs.

In 1989 the Spineless Tagless G-machine (STG) was introduced by Peyton Jones [28]. It is based on the G-machine and the abstract machine Tim [29], but focuses on the use of closures as opposed to an explicit representation of the graph, leading to a further decrease in bookkeeping and improved mapping to traditional processors. At the time of writing, the Haskell compiler GHC [30] uses an STG-based approach.

1.2.6 Combinators return

In addition to their continued use in VossII to implement the language fl, two re-emergences of fixed combinators for evaluating functional programs should be

mentioned. The first is MicroHs [20] (also discussed in Section 1.1), a small and portable Haskell compiler written by Lennart Augustsson (one of the creators of the G-machine). Although described as a toy project, the compiler supports much of the Haskell 2010 standard [31], and is self-hosting. The second is Lambda-One [32], a hardware architecture for running functional programs that will be discussed more in the next section. Both make use of the simplicity and power of fixed combinators; in MicroHs for the sake of a small compiler and runtime, and in Lambda-One (presumably) to simplify the hardware.

1.2.7 Hardware for functional programs

A number of architectures specific to pure functional programming have been developed, in the decade following Turner's 1979 paper [23] on the use of combinators for program evaluation.

By the 1990s interest seems to have waned, likely due to the arrival of abstract machines (such as STG) that work well on traditional architectures, which were experiencing rapid gains in speed at the time. In recent years, however, interest has renewed. There are several plausible reasons for this: slowed growth of single-thread performance, increased interest in accelerators, new priorities in domains such as IoT, availability of reconfigurable hardware such as FPGAs, and accessibility of high-level design tools and languages. Several such (comparatively) recent projects are discussed below.

The Reduceron [33] is a processor for functional programs designed to be used on FPGAs. By using wide words and extensive parallelism, it can perform substantial reduction steps in a single cycle. The program is stored in a form conducive to this. While three to four times slower than a 3GHz PC running binaries compiled by GHC, this is largely a product of the Reduceron's lower clock speed on the FPGA (96Mhz). The authors' estimates suggest that the Reduceron would compare favorably in run time to an FPGA soft-processor (MicroBlaze) running GHC-compiled binaries by an entire order of magnitude. Energy measurements are not given, however, and the parallelism used by the Reduceron suggests a relatively high power consumption. Additionally, the design relies on fast memory access, which is threatened in the case of off-chip RAM; the authors suggest either special types of RAM or the use of caches/buffers.

Lambda-One [32] is a platform for embedded functional programming, with a combinator-based graph reduction processor (Blackbird). It is realized as an FPGA on an printed circuit board, complete with external RAM and a wide array of peripherals. The CPU uses several refinements to accelerate the graph reduction process. However, its performance is unclear in relation to a software implementation of graph reduction on an ordinary processor, as is its energy consumption. It also uses a stop-the-world garbage collector, which creates negligible latency in the on-chip memory but appears likely to create long delays when applied to the 512MB off-chip memory.

Heron is a recent FPGA-oriented graph reduction processor, with source code [34] available as of February 2024 and a forthcoming publication. A presentation given at HAFDAL 2024 [35] indicates that the processor is similar in design to the Reduceron, extending it with further performance-enhancing features. The power consumption appears to be at least 1W, considerably lower than desktop computers but one or two orders of magnitude higher than commercial low-power processor cores such as the ARM Cortex M0+ [36] (albeit at a considerably greater speed), limiting use in

ultra-low-power IoT devices.

1.3 Hardware design

In order to design computer hardware one typically uses a programming language, called a *hardware description language* (HDL), to specify the structure or behavior of a component or system. The source code is then passed through a series of tools that produces circuit descriptions at increasing levels of detail and decreasing levels of abstraction, eventually culminating in a manufacturable design.

HDLs come in many varieties with differing programming models and levels of abstraction. Since the introduction of High-Level Synthesis (HLS)—where the designer describes the hardware at a high abstraction level, and a compiler generates a circuit with (hopefully) the same behavior—there has been a tension between the level of abstraction and the quality of the resulting hardware: a higher level of abstraction makes life easier for the designer, but gives less control and can lead to worse performance of the final design. This section will briefly discuss several hardware description languages (and classes thereof) and the programming models they offer.

Traditional HDLs such as Verilog [37] and VHDL [38] offer a low level of abstraction. Specification can be structural or behavioral, but the behavioral parts are fairly limited, and not all parts of the languages are synthesizable by popular tools. Due to their ubiquity, these languages also serve as target languages for other tools such as higher-level HDLs.

High-Level Synthesis typically refers to commercial tools that turn C or C++ programs into circuits, such as Vivado HLS [39] and Cadence Stratus [40]. By offering a high level of abstraction for the behavior of a process—especially with scheduling—these can make the process of designing hardware or exploring different design alternatives significantly simpler and faster. The results can vary greatly in quality, however, limiting adoption [41]. Bugs also abound in HLS tools [42], potentially hampering design and decreasing confidence in the resulting hardware. A more subtle disadvantage is that the programming model does not lend itself to composition: while one can call functions inside a module—a process requiring passing both data and control back and forth—a similar interaction with an external piece of hardware is not given first class status and needs to be carried out manually, encouraging a more monolithic design.

Approaches based on functional programming often bring the compositionality that the paradigm is known for. VossII [25], Lava [43], Blarney [44], and Chisel [45] provide frameworks for describing hardware using fl/hfl (VossII), Haskell (Lava and Blarney) and Scala [46] (Chisel). CλasH [47] is a Haskell-like language for describing hardware directly, rather than through an embedding. None of these other than Blarney offer a direct way to write blocking, algorithm-like processes. Despite working primarily at the register-transfer level (RTL), Blarney includes a higher-level imperative EDSL that modules can be written in.

BlueSpec [48] offers an entirely different approach to hardware design, prioritizing modularity and compositionality, and using a very elegant abstraction: *guarded atomic actions*. The latter specify an atomic event and a precondition for its occurrence. Although it satisfies the surprisingly rare combination of being elegant and able to express a pipelined processor, this model has a drawback: procedures that take

multiple clock cycles (e.g. non-trivial algorithms) need to be broken down into these atomic rules, obfuscating control flow and requiring the designer to make scheduling decisions.

Spade [49] is an HDL that incorporates niceties from modern languages such as a proper type system and pattern matching, and supports pipelining as part of the language. Scheduling and stall/bypass logic are manual, limiting the level of abstraction, but can arguably be expressed more easily and clearly than in traditional HDLs.

PDL [50] is a language for describing pipelined, speculative processors while keeping one-instruction-at-a-time semantics. The separation between stages is manual; arguably a desirable lack of abstraction when designing processor cores, but the compiler performs the typically complex task of verifying that a design satisfies one-instruction-at-a-time semantics. Primitives for resource acquisition and speculation abstract away details of this logic. This makes a remarkable departure from the usual abstraction-quality dichotomy: it seems to truly both have its cake and eat it. Although not strictly confined to processors, the computation model does appear to be somewhat limited in domain.

1.4 Research and contributions

This thesis aims to answer two research questions. First, whether in a low-energy context (such as IoT) a small, combinator-based microprocessor architecture is a viable approach to running lazy functional programs. Exploration of this led to a second question: whether a comparatively minimal high-level hardware design language, with an emphasis on modularity rather than feature-completeness, is an effective tool to create hardware of the scale and variety under consideration. Both questions share a common theme: the potential value of simplicity.

Three artifacts were ultimately developed to address these questions: Cephalopode, a combinator-based graph reduction processor; Stately, an editor for finite state machines; and Bifröst, a language for hardware design.

1.4.1 Cephalopode

Cephalopode, described in its latest revision in Paper A and in its infancy in Paper B, is a microprocessor architecture for functional programs implemented using the hardware design and verification platform VossII [25], and the aforementioned Stately and Bifröst.

The processor aims to be as low energy as possible while still meeting several other requirements aligned with an IoT context: low latency, in particular with regards to garbage collection; safety, in both memory and arithmetic; and suitability for multitasking. The main criterion for success—indicating viability of a combinator-based approach to functional programming on IoT—is that the processor's performance match or exceed that of a traditional processor running the same functional program. Here "same functional program" refers to the source program—modulo minor differences from the language/environment—not the resulting machine code, which should be tailored to suit each architecture. The processor's performance must also be in some sense *reasonable* relative to a more conventional, non-functional implementation such as native C code that performs the same task; this criterion is more fuzzy due to the subjective valuation of language abstraction level and safety versus

program performance (a topic that still incites arguments to this day). For both criteria the notion of "performance" refers primarily to energy consumption for a given task, provided that the speed is in an acceptable range for an IoT platform. Speed itself is considered an added bonus. The remainder of this section will give an overview of Cephalopode and its evaluation relative to these goals.

Regarding tooling, Cephalopode is implemented using the VossII platform. While other platforms and languages offer compelling programming models perhaps more amenable at then outset to the processor's implementation, VossII was chosen for its integrated design and verification capabilities. Specifically, long-term plans for Cephalopode include refinement of its design using provably correct transformations, and verification of important properties such as garbage collector soundness and memory safety. Stately and—later and more extensively—Bifröst were used to bridge the higher-level needs of many parts of the processor design to the comparatively low-level RTL circuit model used by VossII, while preserving the integrated design and verification benefits of the latter, offering a degree of integration, and avoiding compatibility difficulties such as irreconcilable type systems. While the hardware generated by Bifröst is not optimal (see Chapter 6), the tool was still used extensively for several reasons: *(i)* to make the design and design space exploration process more manageable, *(ii)* to reduce the risk of subtle bugs, *(iii)* to make use of its automatic clock gating in order to reduce power consumption, *(iv)* in anticipation of future work on the compiler (e.g., better circuit generation, more aggressive power management features), and *(iv)* to serve as a "golden model" for refinement using VossII, which would likely be necessary to generate optimized hardware even if a different language were used.

In the Cephalopode architecture programs are expressed as graphs built from a family of combinators, and the processor carries out graph reduction directly, i.e. without any software interpreter. The set of combinators supported is an extension of that described by Peyton Jones [24]. It is primarily a fixed set, though it contains two combinators C_n and L_n with a numerical index, whose reduction rules are as follows:

$$C_n\, f\, e_1\, \ldots\, e_n\, x \Rightarrow f\, x\, e_1\, \ldots\, e_n,$$

$$L_n\, e_1\, \ldots\, e_n\, x \Rightarrow x\, e_1\, \ldots\, e_n.$$

Normally the strict nature of operators requires recursive evaluation of their arguments (e.g. first evaluating `e1` and `e2` in order to evaluate `(+) e1 e2`), but this is avoided using a clever trick taken from the Reduceron: a reduction rule $v\, f \Rightarrow f\, v$ is added for all non-function values v, and the compiler turns strict functions inside out (e.g. `(+) e1 e2` becomes instead `e2 (e1 (+))`). This works for saturated applications, but not for partial ones, where instead the operator needs to be replaced by a function that carries out this reordering dynamically; this is facilitated by the L_n combinator.

While I/O and multi-tasking primitives have not yet been implemented, the processor contains primitives for integer and boolean arithmetic and list processing. Arithmetic is variable-precision: integers may be stored in a single graph node or a linked list of graph nodes. The maximum size is bounded by that of the arithmetic unit's local memory (compared to all available memory, as with arbitrary-precision), though a pending architectural change is to spill into main memory when the local one becomes full. A benefit of this approach is that even if extraordinarily large numbers are not needed, the processor can be sized based on the *average case* of integer size rather than the *worst case*, without incurring the overhead of a software

arbitrary-precision implementation. As a result, this prevents integer overflow bugs in all but the most extreme cases.

Although multi-tasking is not implemented, Cephalopode is well suited for it: graph reduction does not use a stack (instead, it stores a pointer back up the graph in each traversed node, at the expense of size), which means context switching is a matter of pausing graph reduction and swapping several pointers. This is especially beneficial on an embedded system, where quickly servicing interrupts is often important. Related to the desire to avoid an evaluation stack, in order to avoid recursive evaluation a clever trick from the Reduceron is used to force the evaluation of appropriately-typed expressions.

Garbage collection is given special attention in Cephalopode. Not only can it not be implemented in software by the nature of a graph reduction architecture, but doing so in general comes with a significant overhead. As discussed earlier, this is especially the case when a pauseless garbage collector is used, as opposed to a naïve stop-the-world one. In order to avoid long delays—which could be pronounced and problematic on an embedded device—Cephalopode implements a garbage collector in hardware that runs in parallel to the program. Rather than using write barriers (which require an additional read for each memory write) to ensure soundness, Cephalopode's garbage collector operates on a snapshot of a coherent state of the graph. Taking this snapshot by duplicating the entire graph would result in significant latency, so instead the snapshot is taken lazily: memory writes after the moment the snapshot is nominally taken are redirected to alternate locations, leaving the old graph intact. This does require an extra read prior to every write in order to know which of the two possible locations an address is associated with should be used (and even an extra write to update this information), but these metadata bits can be stored in a high-speed, on-chip memory, meaning that this read can be performed combinationally or with a very small delay compared to ordinary off-chip memory. A garbage collection unit runs an ordinary mark-and-sweep algorithm over the snapshot, and any nodes that were garbage in the snapshot are safe to free despite the graph having evolved since (the soundness of this is argued by Yuasa [51]).

Downsides of the garbage collection system are almost entirely in memory elements and sizes: the requirement that physical memory be twice the size of usable memory in order to maintain the snapshot, the need for high-speed metadata memory to facilitate the snapshot, the need for a stack memory to store as many pointers as possible nodes in memory, and the need for a memory to store mark bits. While these are quite dramatic, alternatives are not particularly appealing. A generational garbage collector [52] would require greater complexity and bookkeeping to track pointers from older generations to younger ones—the latter likely requiring additional writes to main memory as opposed to an on-chip memory—and usually require long pauses when tracing older generations (though this can be avoided [53]). In order to satisfy the soundness criteria described by Yuasa, ordinary pauseless and incremental garbage collector designs (generational or otherwise) evacuate pointers before they are overwritten for the first time during a marking phase, necessitating an extra read from main memory prior to each such write, as well as a write to the garbage collector's stack; this approach would avoid the costly doubling of main memory for the snapshot, but with an increase memory traffic and likely both time and energy consumption as a consequence. The simultaneous access to the garbage collector's stack from both the garbage collector and the reduction engine during pointer evacuation also requires synchronization, increasing complexity and

potentially introducing further delays.

Graph reduction sometimes creates nodes called *indirection nodes* that simply serve as aliases of other nodes. The ratio between indirection and non-indirection nodes is unfortunately not bounded, meaning that eventually indirection nodes can occupy virtually all of memory. To lessen this, the sweep phase of the garbage collector additionally performs an operation similar to *path compression* from disjoint-set data structures on chains of indirection nodes in the graph. This does not immediately remove any nodes—a difficult task when accessing memory concurrently to the main program—but ensures that no indirection node points to another one, i.e. there are not any chains of two or more indirection nodes in a row. This in turn implies that there can only be one live indirection node per pointer per non-indirection node, and others will be collected on the next garbage collection pass. If the program creates indirection nodes at a negligibly slow speed relative to how often garbage collection is run, this bounds the ratio at 2:1; unpleasant but manageable. Methods for bypassing all indirection nodes without causing significant delays in the program or garbage collector remain to be explored.

Cephalopode was synthesized and evaluated in comparison to a publicly available RISC-V core [54]. Both were synthesized using the using Cadence Genus 18.14 and ASAP7 PDK [55], a 7nm predictive process design kit and standard cell library.

For the RISC-V core, code was generated using MicroHs [20] (for functional programs) and GCC [56] (for native C implementations, and also used in the backend of MicroHs). MicroHs was chosen as it is the only known way to run lazy functional programs on microcontroller-sized processors; a comparison to GHC [30] would be preferable as the latter is more representative in its performance, but unfortunately it is not yet able to emit code for RISC-V. Instead, a speculative, back-of-the-envelope comparison with GHC was performed based on the relative performance of MicroHs and GHC on a desktop computer; for a given program P we use the approximation:

$$\text{Energy}_{P,\text{GHC,RISC-V}} \approx \text{Energy}_{P,\text{MicroHs,RISC-V}} \times \frac{\text{Instructions}_{P,\text{GHC,x86-64}}}{\text{Instructions}_{P,\text{MicroHs,x86-64}}}.$$

The instruction counts were determined using Valgrind [57]. This approximation is carried out with and without GHC optimizations enabled: this was chosen since many of the optimizations that occur early in the GHC pipeline are likely applicable to Cephalopode as well, which currently uses a fairly naïve compiler.[1] Given the resource constraints in an IoT device it is quite possible that GHC's complex runtime would need to be simplified at the cost of performance, so the estimates for it may be optimistic in this regard.

A selection of benchmark programs were compiled for the platforms, and the resulting hardware-and-ROM combinations were simulated at 100MHz in order to obtain running time, estimated energy consumption, and memory access counts. The programs, partially based on a small IoT-oriented set [58], consisted of a mixture of basic arithmetic, matrix manipulation, and list processing, as well as an implementation of a small neural network. These were chosen due to their applicability to IoT devices, particularly those that process sensor data. The resulting relative energy, running time, and memory accesses are shown in figures 1.3, 1.4, and 1.5.

At equal clock frequencies (100MHz), Cephalopode appears to run approximately

[1] Since we are speculating about future developments of GHC here, it seems fair also to consider improvements to Cephalopode's compiler.

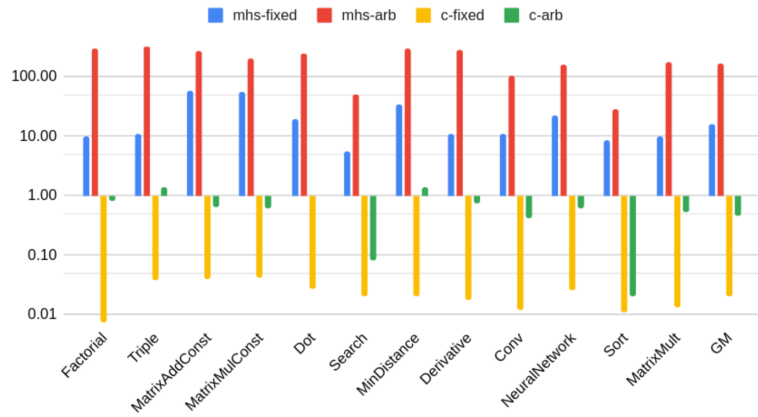

Figure 1.3: Simulated energy expenditure relative to Cephalopode.

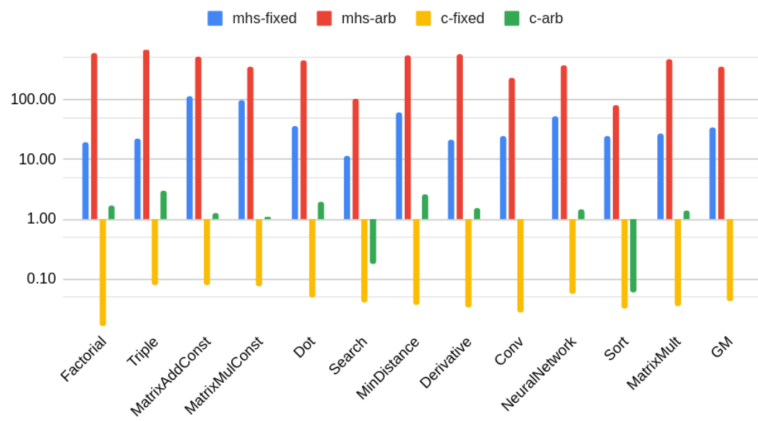

Figure 1.4: Simulated running time relative to Cephalopode.

30 times faster than MicroHs on RISC-V using fixed-precision arithmetic,[2] and consumes about 16 times less energy.

The speculative comparison to GHC (Figure 1.6) indicates that Cephalopode uses approximately the same amount of energy as unoptimized GHC for RISC-V would when the latter uses fixed-precision arithmetic, and half as much energy when it uses arbitrary-precision arithmetic. GHC's optimizations—some likely applicable to Cephalopode—result in it being estimated to use 3 times less energy than Cephalopode with fixed-precision, and equal with arbitrary-precision. The relative running times match the relative energy usage but with an additional factor of two in favor of Cephalopode, meaning that Cephalopode is expected to take less time in all cases except for optimized GHC with fixed-precision integers.

When comparing to native C code that uses ordinary fixed-width integers, Cephalopode uses about 50 times as much energy and 23 times as much time, appar-

[2] Arbitrary-precision arithmetic adds a factor of around 25x to MicroHs in both time and energy for implementation reasons, even if all integers are small; benchmarking of GHC on x86-64 suggests this can be reduced to around 2x.

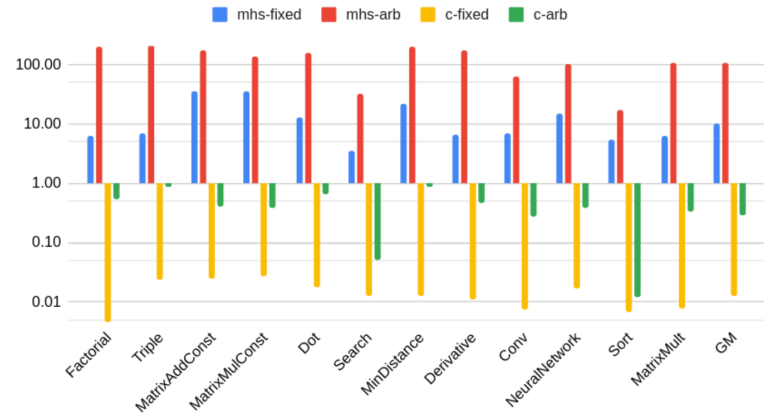

Figure 1.5: Number of memory accesses relative to Cephalopode.

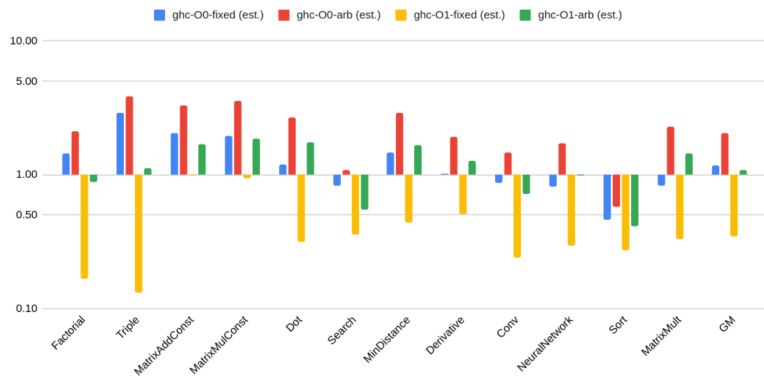

Figure 1.6: Estimated GHC energy expenditure relative to Cephalopode.

ently a result of its significantly higher memory traffic. This is not ideal, although it is not far from the sacrifices made for JavaScript through the V8 engine, and Python through CPython: one study found that these respectively incur an 8x and 29x slowdown over C++ [59], suggesting that Cephalopode may in fact be more performant as far as speed is concerned than CPython. When the native C code uses an arbitrary-precision library[3] to match the arithmetic safety guarantees of Cephalopode, the performance differences shrink dramatically: Cephalopode uses on average twice as much energy and the same running time as the native C implementation, while offering a much higher level of abstraction to the programmer.

To summarize, while Cephalopode is unlikely to compete with a fixed-precision C implementation, it offers a much higher level of safety and abstraction for nearly the same price as adding arbitrary-precision code alone. It is over an order of magnitude more performant than existing lazy functional programming implementations for IoT-scale processors (MicroHs), and appears competitive with hypothetical ones (GHC). As a result, one may conclude that Cephalopode successfully meets the goals and criteria described at the beginning of the section.

[3]In our benchmarks, the one from VossII.

1.4.2 Stately

Stately, described in Paper C, is a visual editor for finite state machines, providing a graphical interface for creating, editing, and testing them. In addition to providing a more intuitive view than a textual (code) representation, it provides two primary benefits: *(i)* separation of all logic by state, and *(ii)* a mechanism for making states that are transitioned to immediately, rather than on the next clock cycle.

The first means that next-state logic and output logic are defined inside of each state, not globally.[4] This is done by giving each state its own source code, appearing in a text field in the editor when the state is selected. A state that initiates a memory read could be programmed as follows:

```
if not mem_ack:
    emit mem_req
    goto reading
```

If the `mem_ack` signal is high, the machine remains in the same state, and if it is low then the `mem_req` signal is raised (for the current clock cycle) and the machine transitions (on the next cycle) to the state named `reading`.

This matches how we usually think about state machines during their design, namely *what happens in each state* rather than *for which states each thing happens*. As a result, the behavior of the state machine is clearer, and easier to modify without introducing errors.

The second, dubbed *virtual states*, are states that will not exist in the compiled hardware because they are instead absorbed into previous states in a manner similar to inlining a function. Formally, if a state A transitions to a virtual state B if condition c holds, then all of B's behavior will occur in A, predicated by c. A specific example is given in Listing 1.1. While quite simple, this solves a frequent dilemma: whether to make a state machine that is clear and maintainable but uses more states, or one with fewer states but convoluted or duplicated logic. By making states virtual, they can be folded together with other states at compile time, getting the best of both worlds.

```
-- Regular state A
if foo:
    emit x
    goto B

-- Virtual state B
if bar:
    emit y
    goto C
else:
    goto D

-- Effective behavior of A
if foo:
    emit x
    if bar:
        emit y
        goto C
    else:
        goto D
```

Listing 1.1: Example of virtual state behavior.

[4] For certain signals this may be undesirable, so Stately also allows global signal definitions such as defining `is_reading` to equal `not rw`.

This is especially true for bottlenecks in an FSM, where N states all transition to a single state S, which based on some conditions transitions to one of M states. It might be desirable to skip S in order to save a clock cycle, but with an ordinary FSM this would turn the $N + M$ transitions of the original machine into a substantially greater $N \times M$ transitions, complicating it and making maintenance a challenge.

While not too dramatic of a departure from existing models of computation, Stately proved quite useful in the early development of Cephalopode. More specifically it was used for the control logic of the first versions of the graph reduction engine and arbitrary-precision divider, among other components. Both of these are quite substantial, and having created the former FSM, the author suspects that it would have been significantly less efficient or taken far longer to construct without Stately and virtual states in particular. The visual nature of the editor was also appreciated; while not amenable to very tiny nor enormous state machines, in the case of those mentioned previously it made them organized and readable at a glance, and by making transitions visually explicit it helped avoid careless mistakes that could otherwise go unnoticed.

1.4.3 Bifröst

Bifröst, described in Paper D and elaborated upon in Chapter 6, is a language for designing modular hardware at a high level. It arose from the frustration of separately developing the datapath (in hfl) and control logic (with Stately), and the observation that many of the more difficult to implement parts of Cephalopode had behavior resembling "algorithmic" imperative programs. The sequential nature of this behavior meant that most gains from parallelism within these modules—at least in the designs under consideration—would come from tedious packing of operations into a single cycle, rather than complex re-ordering or clever tricks. A language—Bifröst—was created that could express these operations, along with a compiler to translate such programs into RTL circuit descriptions in hfl. Development continued in parallel to Cephalopode, the latter serving as both a case study for evaluation and a source of inspiration for functionality and design decisions.

The motivation for Bifröst is discussed in more detail in Chapter 6, but its goal can be summarized as follows: to create a language with an algorithm-friendly, high-level semantics that is suitable for use for small modules in a diverse environment; and in doing so to create a higher-level starting point for refinement and verification in VossII. Since compiler implementations are often improved over long time frames, the success of Bifröst in the context of this thesis mainly concerns its design—does the programming model it provides realize the goals above? This can be divided into two criteria: *(i)* whether the programming model is well-suited from the programmer's perspective to describing hardware of the variety it aims to, and *(ii)* whether it is possible to generate circuits of acceptable quality from these descriptions. As with Cephalopode, the subjective valuation of abstraction level and performance precludes a purely objective judgement, however through the quantitative and qualitative evaluation both in this section and in greater detail in Chapter 6 we argue that Bifröst successfully meets both criteria described above.

Syntax quirks and not-yet-implemented features aside, Bifröst largely resembles an ordinary imperative language: there is a `main` function, variable declarations, assignments, branching, loops, return statements, function calls, etc. Its semantics are sequential; each statement nominally runs before the next. There are, however,

language aspects that differentiate it from both ordinary imperative languages and from other HDLs. Unlike the former, *(i)* it is compiled to a hardware module rather than to software, precluding behavior such as dynamic memory allocation; *(ii)* it includes two different notions of function calls, one ordinary and one for I/O operations; and *(iii)* it may instantiate and run other modules. Unlike the latter, *(i)* the circuit is defined in terms of a `main` function, rather than continuous behavior; *(ii)* it is strongly and statically typed, but details such as the sizes of types are not declared; and *(iii)* all I/O takes place through function call-like constructs called *actions*, rather than setting values of signals.

Actions are arguably the defining feature of Bifröst. They are, in essence, external function calls to other hardware components (including instantiated children), the idea being to bring the semantic benefits of function calls to hardware. They are unrelated to the notion of actions in Blarney, and differ from the guarded atomic actions in BlueSpec in that they may block and how they are used. By merging the act of transmitting data to another component and receiving a response back, not only is programming simplified but the language can operate at a higher level of abstraction. A simple example of an action invocation follows:

```
temp = do read_temperature SENSOR_OUTSIDE;
```

Information flows out through the argument `SENSOR_OUTSIDE` and (at some point) back in through the returned temperature.

There are two main ramifications of this abstraction, both arising from the compiler's understanding of exactly how and when communication is occurring. First, the compiler can easily synthesize logic for a handshake protocol for the communication. Returning to the example above, the declaration of `read_temperature` could indicate that a four-phase handshake is to be used when communicating with the temperature sensor controller. This protocol flexibility makes it easy to interact with a wide variety of hardware modules not written in Bifröst, facilitating integration. Furthermore, protocols in Bifröst can include signals not only for data transfer but also for power management, allowing Bifröst to synthesize clock gating[5] circuitry for modules written in Bifröst, and control that of other (compatible) modules that are called via actions. Generating this logic is an error-prone activity, so having the compiler generate it at the outset—when the information about control flow is available at an appropriate abstraction level—avoids the risks and difficulty of retrofitting it later.

Second, with some cooperation from the user Bifröst can safely parallelize parts of the program without sacrificing its sequential semantics, even taking side-effects of I/O operations into account. The cooperation needed from the user is to declare these side-effects: the outside world is characterized based on a finite set of *aspects*, and for each action, the aspects the outside world it depends on (or *reads*) and the aspects it modifies (or *writes*) are listed. These aspects are arbitrary atoms such as `STACK` or `MEMORY`; the compiler does not need to understand what they represent, but they do need to reflect the relationships of the I/O operations in question. Provided these are defined appropriately, the compiler can determine whether running two actions at the same time is guaranteed to have the same effect as running them in sequence, i.e. whether they may be parallelized without changing the program's semantics.

[5] A way to dramatically reduce power consumption by temporarily disabling the clock signal in dormant parts of a circuit.

Scheduling makes an assumption that appears to hold in Cephalopode: that in most cases, the work done by combinational logic is trivial glue between invocations of actions (such as RAM access). Bifröst is therefore quite aggressive when scheduling combinational logic, essentially treating it as free. What guides scheduling instead are dependencies between action invocations (one depending on the results of another, or a pair being unsafe to parallelize), and manual cycle-breaks inserted by the user. To keep scheduling predictable and relatively stable, the scheduler does not make any attempts to re-order statements; it simply packs as much into the current cycle as possible and begins a new one as needed.

A notable downside of this model—imperative programs that invoke dynamically-timed actions—is that it is not amenable to pipelining. Although refinements could be made to introduce pipelining in some situations, it is not clear how this would be done generally and circuits created by the Bifröst compiler currently resemble those of multi-cycle processors in their operation.

The main evaluation of Bifröst has come through the development and evaluation of the Cephalopode processor. The first version of Cephalopode used Stately for much of the control logic, the datapath and remaining control logic being written by hand in hfl. Soon after Bifröst's introduction, the (complicated) arbitrary-precision divider was re-implemented in it, and it became apparent that this was advantageous from exploration and maintenance perspectives. When several major architectural revisions to Cephalopode were planned, it was decided to write a second version of the processor using Bifröst wherever possible. Although it did not fit everywhere, and sometimes required some contortion,[6] it comprises a substantial proportion of Cephalopode and made the design process faster and more flexible, allowing significant changes to be made in a matter of hours rather than weeks. In addition to the benefits on the programming side, the performance of Cephalopode with respect to energy, computation speed, and timing reveals that the hardware generated by Bifröst is (after synthesis) sufficiently performant for real-world use cases.

Direct comparisons (detailed in Chapter 6) between hardware created by hand in hfl and using Bifröst indicate that the latter carries a performance penalty, but—at least in the cases tested—only a moderate one. While determining the interaction between the hardware patterns used by Bifröst and the refinement capabilities of VossII is beyond the scope of the research in this thesis, it is hoped that refinement would lessen the performance difference, while preserving the benefits offered by Bifröst. The direct comparisons also corroborate the experience developing Cephalopode in terms of Bifröst's programming model; the latter proved to be quite advantageous for describing the algorithmic computations it was envisioned for.

1.4.4 Conclusion

Through these three artifacts, this thesis claims to provide an affirmative answer to both research questions: in a low-energy context, custom architectures for functional programming—including combinator machines—are shown not only to be viable, but competitive today; and small, modular, high-level hardware design languages are demonstrated to be extremely useful for the realization of novel architectures. Furthermore, the exploration carried out suggests several avenues for further research.

[6] The memory controller was a good candidate for Bifröst due to its complexity, but had to be reformulated to work with Bifröst's single-entry-point program model.

1.5 Future work

1.5.1 Low-power hardware for functional programs

Cephalopode itself has many improvements that can be made, first and foremost the addition of I/O and multi-tasking operations. Possible gains in performance gains may be had through alterations to the memory structure—in particular, the trade-offs related to garbage collection may benefit from further experimentation—and through a plethora of local refinements. A very interesting experiment would be to port GHC to Cephalopode as well, to obtain a both fair and state-of-the-art comparison.

Regardless, it is clear that if a hardware implementation alone is enough to resurrect a graph reduction strategy largely superseded over thirty years ago, perhaps one ought to investigate applying similar techniques to other strategies that have less of an interpretative overhead. The Reduceron—and more recently, Heron—indeed do so with template code. The former prioritizes speed (as opposed to energy), and both would appear to be constrained as FPGA-centric designs. A hybrid between this approach and that of Cephalopode seems promising. Alternatively, one of the abstract machines described in Section 1.2.5 may also be a good suitable candidate for low-power hardware implementation.

Finally, verification of desirable properties of the design, such as memory safety, correctness of the garbage collector, and process isolation seem an appealing way to leverage the design's comparative simplicity and implementation using VossII.

1.5.2 Hardware design

Working with hardware description languages it is nearly impossible not to come to the conclusion that there is a glaring semantic gap. Languages that excel in description of parallel happenings seem to be lousy for programs that are (mostly) sequential in nature, especially with complex control flow; the forest is not seen for the trees. On the other hand, languages that express the latter types of processes succinctly (like Bifröst in some aspects, though not all) appear miles away from the hardware, and handle important performance concerns such as pipelining poorly if at all. And, most concerning, there are very few languages that can be considered to be anywhere in between.

Thoughts on this issue are often expressed in relation to high-level synthesis: naysayers view the effective translation from behavioral to structural as intractable, whereas those with more optimism anticipate smarter, better HLS systems that can accomplish it as well as the translation from Haskell to machine code. Both of these, however, miss the point: we seem to want to express computation somewhere on a continuum, but we only have the endpoints.

Thankfully, this seems to be undergoing change: languages like BlueSpec [48] and PDL [50] offer models that begin to blur the distinction (or at least give a radically different view). It could be that the trade-off is fundamental, and that somewhere along the scale of abstraction there is an unavoidable leap. It is also possible, however, that there as an as-yet-undiscovered model that sits somewhere in the middle, and makes itself useful across a good portion of the abstraction spectrum. This would be a very exciting direction to explore in future, Bifröst-esque endeavors.

1.6 Reading this thesis

The remainder of the thesis consists of four papers (A through D), and one supplemental chapter. Paper A is the most recent, and describes the completed Cephalopode processor. It largely supersedes the design described in Paper B, although the latter contains some additional details on the snapshot memory and garbage collection. Paper C describes Stately. Paper D and Chapter 6 describe Bifröst: the language in the former, and the compiler and evaluation in the latter. The author suggests that a hurried reader might most enjoy Paper A and Paper D.